COPENHAGEN
INTERNATIONAL
SOFTWARE DEVELOPMENT
CONFERENCE 2014

goto; conference

USING TYPELISTS AND TEMPLATE METAPROGRAMMING TO GENERATE CODE AND AVOID DUPLICATION

Jesper Clausen
Saxo Bank A/S

Using typelists and Template Metaprogramming to generate code and avoid duplication

Template Metaprogramming in practice

By Jesper Clausen (jcl@saxobank.com)

Solution Architect in Saxo Bank.

SAXO BANK

Teaser (1)

- Imagine having a class that can contains caches for a number of types.
- Every time a cache for a new type is added, some extra lines of code need to be added here and there.
- For each type we instantiate a cache:

```
…
Cache<PointRecord> m_pointCache;
Cache<RectangleRecord> m_rectangleCache;
Cache<CircleRecord> m_circleCache;
Cache<LineRecord> m_lineCache;
Cache<PieRecord> m_pieCache;
Cache<TriangleRecord> m_triangleCache;
…
```

Teaser (2)

- We implement a get and a set method:

```
…
bool get(const Id_t id, PieRecord & record) const
{ return m_pieCache.get(id, record); }

void set(const PieRecord & record)
{ m_pieCache.set(record); }

bool get(const Id_t id, TriangleRecord & record) const
{ return m_triangleCache.get(id, record); }

void set(const TriangleRecord & record)
{ m_triangleCache.set(record); }
…
```

Teaser (3)

- We make yet another case in a switch statement:

```
switch(recordTypeId)
{
case RecordTypeIds::Point:
  return m_pointCache.erase(id);
case RecordTypeIds::Rectangle:
  return m_rectangleCache.erase(id);
…
case RecordTypeIds::Triangle:
  return m_triangleCache.erase(id);
}
```

- And add an extra line here:

```
size_t eraseAll(const Id_t id)
{
  return m_pointCache.erase(id) +
         m_rectangleCache.erase(id) +
         m_circleCache.erase(id) +
         m_pieCache.erase(id) +
         m_lineCache.erase(id) +
         m_triangleCache.erase(id);
}
```

Teaser (4)

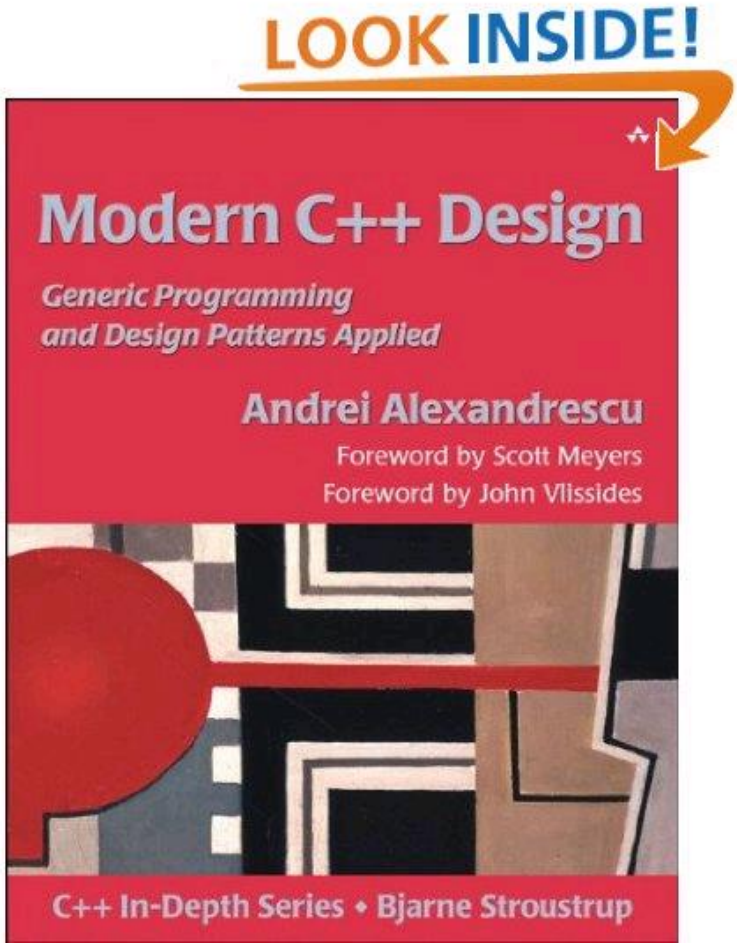
- Supporting a new cache is tedious and error prone. The class contains manually written duplicated code.
- This talk is about how template and typelists can help reducing the task of adding an extra type to adding the type in a typelist definition:

```
typedef Typelist<PointRecord,
        Typelist<RectangleRecord,
        Typelist<CircleRecord,
        Typelist<PieRecord,
        Typelist<LineRecord,
        Typelist<TriangleRecord,
        NullType>>>>>> Geometry2DTypelist;
```


SAXO BANK

Background reading

- Typelists are described in “Modern C++ Design” by Andrei Alexandrescu.



SAXO BANK

The Ladder of Evilness

- "All C++ features are evil but some are more evil than others".
- When solving a problem, consider using the simplest features of the language before considering the more complex ones. Aim high on the Ladder of Evilness:
  - Reuse of existing code (e.g. STL).
  - Free functions (procedural programming, C/Pascal/…)
  - Free template functions
  - Classical OOP (Classes, interfaces, …)
  - Class templates
  - Template metaprogramming (template specialization)
  - Macros
  - Automatic code generation

  Code duplication (Hell, not a step).
- If the step chosen requires manually written duplicated source code, you fall off the ladder as illustrated!
- Don't go to extremes.

Use Case

- In Saxo Bank we have a specific use case where we use simple structs (PODS) for transporting data between systems.
- The system can serialize and de-serialize such structs and distribute them between systems.
- We have 40+ of these structs.
- Each struct has a unique integer **Record Type ID** assigned to it.
- We have a cache template, which needs to be instantiated for all types.
- Structs that are inserted, updated or deleted in the cache will be distributed automatically. The distribution system is an observer to the cache.
- NOTE: No actual production code will be shown here, only a simplified and less exciting example that demonstrates the important points more clearly.

Example “Data Packages”

```
namespace Model
{
  struct Point
  {
    int X, Y;
  };

  struct Rectangle { ... };
  struct Circle { ... };
  struct Line { ... };
  struct Pie { ... };
  struct Triangle { ... };

  struct Point3D { ... };
  struct Box3D { ... };
  struct Circle3D { ... };
  struct Line3D { ... };
  struct Triangle3D { ... };
}
```

- Simple geometric structs are used in this example.
- There are both 2D and 3D geometric structs.

Records

- The structs are wrapped in **Record**s.
- The record type ID (**RecordTypeId**) and the type of the data package (**DataPackage_t**) are template parameters and **are available at compile time**.
- Each instance of a record contains an **instance** of the data package and its **ID**:

```
template<typename T, RecordTypeId_t RECORD_TYPE_ID>
class Record
{
public:
  typedef typename T DataPackage_t;
  static const RecordTypeId_t RecordTypeId = RECORD_TYPE_ID;
  Record();
  Record(const Id_t id, const DataPackage_t& dataPackage);
  Record(const Record& right);
  Record &operator=(const Record &right);
public:
  DataPackage_t &get() { return m_dataPackage; }
  const DataPackage_t &get() const  { return m_dataPackage; }
  Id_t getId() const { return m_id; }
private:
  DataPackage_t m_dataPackage;
  Id_t m_id;
};
```

Record Types

- The records are typedef'ed so that the relation between the data package type and the corresponding record type ID is available at compile time:

```
// 2D Geometry
typedef Record<Point, RecordTypeIds::Point> PointRecord;
typedef Record<Rectangle, RecordTypeIds::Rectangle> RectangleRecord;
typedef Record<Circle, RecordTypeIds::Circle> CircleRecord;
typedef Record<Line, RecordTypeIds::Line> LineRecord;
typedef Record<Pie, RecordTypeIds::Pie> PieRecord;
typedef Record<Triangle, RecordTypeIds::Triangle> TriangleRecord;

// 3D Geometry
typedef Record<Point3D, RecordTypeIds::Point3D> Point3DRecord;
typedef Record<Box3D, RecordTypeIds::Box3D> Box3DRecord;
typedef Record<Circle3D, RecordTypeIds::Circle3D> Circle3DRecord;
typedef Record<Line3D, RecordTypeIds::Line3D> Line3DRecord;
typedef Record<Triangle3D, RecordTypeIds::Triangle3D> Triangle3DRecord;
```

- These are the supported **record types**.

The Cache

- The sample cache is very simple. It has get, set and erase methods.

```
template<typename T>
class Cache
{
public:
  typedef typename T Record_t;
public:
  bool get(const Id_t id, Record_t &result) const;
  void set(const Record_t &result);
  size_t erase(const Id_t id);
private:
  // Irrelevant details…
  …
};
```

- The cache takes a record type as template parameter and contains instances of that record identified by their ID.
- The cache is a container for the record type.

Instantiating Caches

Our goal:

- To create a **Cache Container** that instantiates the cache for the record types we support.
- To give the user access to the set, get and erase methods of the caches.
- To implement an erase function that can erase all records that have the same ID.

A Cache Container, Take One (1)

- The naïve implementation works fine for just a few caches, but as the number of data packages grow, so does the amount of code duplication.

```
class CacheContainer
{
private:
  Cache<PointRecord> m_pointCache;
  Cache<RectangleRecord> m_rectangleCache;
  Cache<CircleRecord> m_circleCache;
  Cache<LineRecord> m_lineCache;
  Cache<PieRecord> m_pieCache;
  Cache<TriangleRecord> m_triangleCache;
```

- Imagine having 40+ of these…

A Cache Container, Take One (2)

- Get and set methods have to be implemented per type, so it's tedious, duplicated work to maintain this class:

```
bool get(const Id_t id, PointRecord & record) const
{ return m_pointCache.get(id, record); }

void set(const PointRecord & record)
{ m_pointCache.set(record); }

bool get(const Id_t id, RectangleRecord & record) const
{ return m_rectangleCache.get(id, record); }

void set(const RectangleRecord & record)
{ m_rectangleCache.set(record); }
…
```

- The good news is that the compiler will ensure that data is inserted into or retrieved from a cache with the right type.

A Cache Container, Take One (3)

- Erasing is even more painful and the cache to erase from has to be found in a switch statement using the record type ID as type identifier:

```
size_t erase(const RecordTypeId_t recordTypeId, const Id_t id)
{
  switch(recordTypeId)
  {
  case RecordTypeIds::Point:
    return m_pointCache.erase(id);
  case RecordTypeIds::Rectangle:
    return m_rectangleCache.erase(id);
  case RecordTypeIds::Circle:
    return m_circleCache.erase(id);
  case RecordTypeIds::Line:
    return m_lineCache.erase(id);
  case RecordTypeIds::Pie:
    return m_pieCache.erase(id);
  case RecordTypeIds::Triangle:
    return m_triangleCache.erase(id);
  }
}
```

- There is no type safety in this approach. It is easy to erase from the wrong cache by mistake. And a case might be missing.

A Cache Container, Take One (4)

- Erase all with the same id.

```
size_t eraseAll(const Id_t id)
{
  return m_pointCache.erase(id) +
         m_rectangleCache.erase(id) +
         m_circleCache.erase(id) +
         m_pieCache.erase(id) +
         m_circleCache.erase(id) +
         m_triangleCache.erase(id);
}
```

- The developer that maintains the code might forget to erase from a cache or erase from the same cache twice. Both mistakes are in fact done here.
- Imagine having 40+ of these and try to find out if one is missing or added twice.
- The solution does not scale.
- The problem cannot be solved at OOP level on the Ladder of Evilness.
- Dough!

A Cache Container, Take Two (1)

- We can try to move one step down the ladder and use templates.
- Let's solve the problem of having to name each cache instance individually.
- We can create a class template that contains a cache instance of type T as a member:

```
template<typename T>
struct CacheContainerLeaf
{
public:
  typedef T RecordType_t;
  typedef Cache<RecordType_t> Cache_t;
public:
  Cache_t & getCache() { return m_cache; }
  const Cache_t & getCache() const { return m_cache; }
private:
  Cache_t m_cache;
};
```

- The cache container can be modified so that it inherits from multiple instantiations of the class CacheContainerLeaf template for the record types we need to support.

A Cache Container, Take Two (2)

- Multiple inheritance from CacheContainerLeaf instantiations:

```
class CacheContainer :
  private CacheContainerLeaf<PointRecord>,
  private CacheContainerLeaf<RectangleRecord>,
  private CacheContainerLeaf<CircleRecord>,
  private CacheContainerLeaf<LineRecord>,
  private CacheContainerLeaf<PieRecord>,
  private CacheContainerLeaf<TriangleRecord>
{
private:
  template<typename T>
  Cache<T> & getCache()
  { return CacheContainerLeaf<T>::getCache(); }

  template<typename T>
  const Cache<T> & getCache() const
  { return CacheContainerLeaf<T>::getCache(); }
…
```

- In this way, the cache instance for type T can be determined at compile time by type instead of having a distinct name for each cache.
- The cache for type T is retrieved by using the getCache() function(s) which will be optimized away at compile time.

A Cache Container, Take Two (3)

- The get and set methods can be implemented as template functions, so no type specific implementations are needed:

```
template<typename T>
bool get(const Id_t id, T & record) const
{ return getCache<T>().get(id, record); }

template<typename T>
void set(const T & record)
{ getCache<T>().set(record); }
```

  - It is also possible to do a type safe erase template function:

```
template<typename T>
size_t erase(const Id_t id)
{ return getCache<T>().erase(id); }
```

  - This is a big improvement.

A Cache Container, Take Two (4)

- Functions erase and eraseAll still contain duplicated code.

```
size_t erase(const RecordTypeId_t recordTypeId, const Id_t id)
{
  switch(recordTypeId)
  {
    case PointRecord::RecordTypeId:
      return getCache<PointRecord>().erase(id);
    case RectangleRecord::RecordTypeId:
      return getCache<RectangleRecord>().erase(id);
    case CircleRecord::RecordTypeId:
      return getCache<CircleRecord>().erase(id);
    case LineRecord::RecordTypeId:
      return getCache<LineRecord>().erase(id);
    case PieRecord::RecordTypeId:
      return getCache<PieRecord>().erase(id);
    case TriangleRecord::RecordTypeId:
      return getCache<TriangleRecord>().erase(id);
  }
}
```

- The **RecordTypeId** of the record is now used in the switch **case**s.

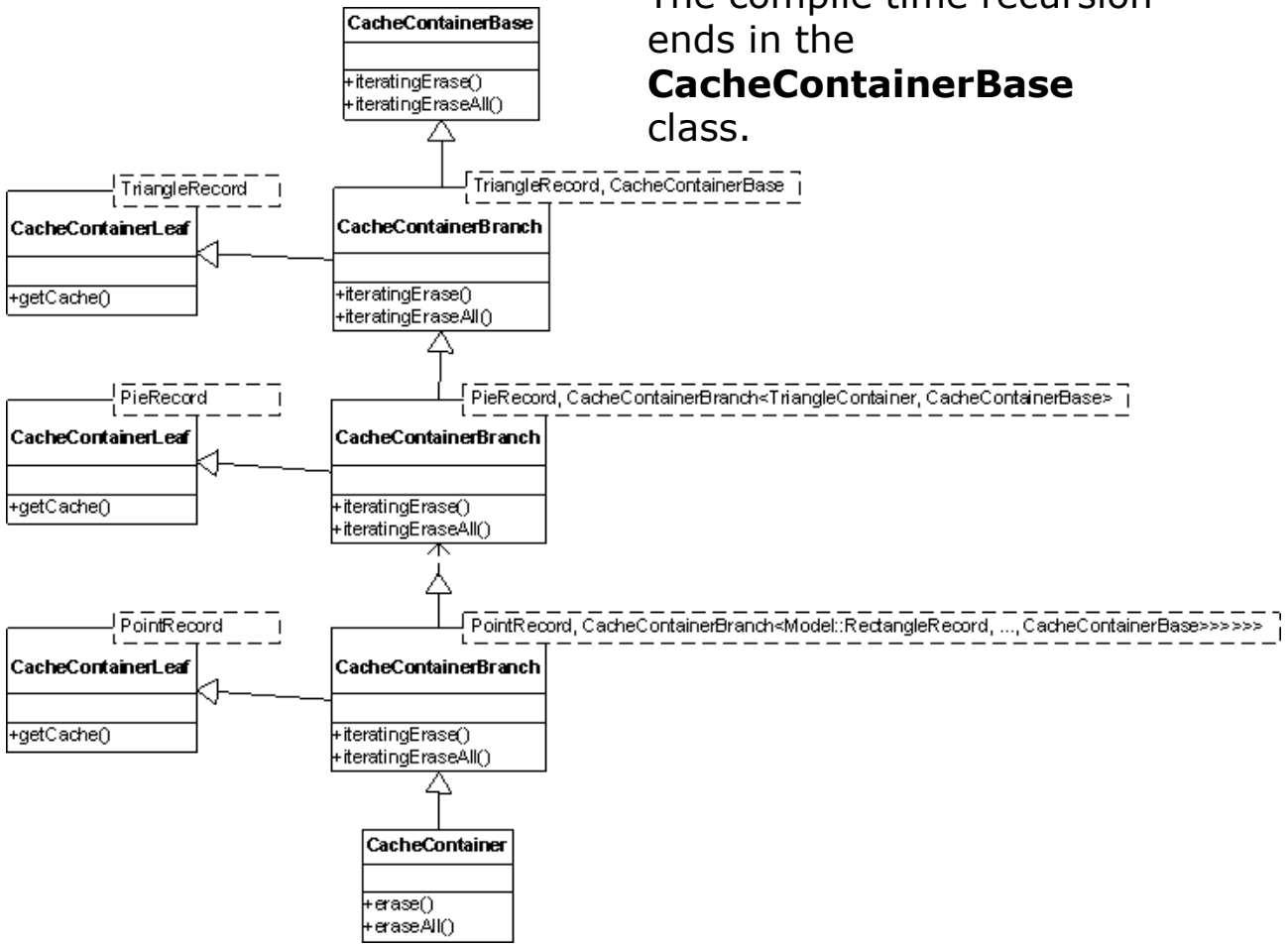
A Cache Container, Take Two (5)

```
size_t eraseAll(const Id_t id)
{
  return getCache<PointRecord>().erase(id) +
         getCache<RectangleRecord>().erase(id) +
         getCache<CircleRecord>().erase(id) +
         getCache<LineRecord>().erase(id) +
         getCache<PieRecord>().erase(id) +
         getCache<TriangleRecord>().erase(id);
}
```

- There is a clear need for a way to **iterate through** the supported record types at runtime.

A Cache Container, Take Three (1)

- The two **iterating** erase functions can be implemented as **compile time recursive functions** in an inheritance hierarchy with one level per type.
- The recursion is done by instantiating the template **CacheContainerBranch** for each type.
- Each branch inherits from both a base class and the **CacheContainerLeaf**. So the multiple inheritance from CacheContainerLeaf is handled as part of the inheritance hierarchy.
- CacheContainerBranch uses the cache of the CacheContainerLeaf it inherits from.



- The compile time recursion ends in the **CacheContainerBase** class.

SAXO BANK

A Cache Container, Take Three (2)

- The **CacheContainerBranch** template takes two template parameters: The type T and a BASE class. The BASE class can either be the **CacheContainerBase** class or a **CacheContainerBranch** instantiation:

```
template<typename T, typename BASE>
struct CacheContainerBranch
  : public CacheContainerLeaf<T>
  , public BASE
{
  size_t iteratingErase(const RecordTypeId_t recordTypeId,
                        const Id_t id)
  {
    if (typename T::RecordTypeId == recordTypeId)
      return CacheContainerLeaf<T>::getCache().erase(id);
    return BASE::iteratingErase(recordTypeId, id);
  }

  size_t iteratingEraseAll(const Id_t id)
  {
    return CacheContainerLeaf<T>::getCache().erase(id) +
           BASE::iteratingEraseAll(id);
  }
};
```

A Cache Container, Take Three (3)

- The base class of the inheritance hierarchy:

```
struct CacheContainerBase
{
  size_t iteratingErase(const RecordTypeId_t,
                        const Id_t)
  { return 0;  }

  size_t iteratingEraseAll(const Id_t)
  { return 0; }
};
```

- The switch statement is eliminated.
- The inheritance hierarchy allows runtime iteration over the supported types.

A Cache Container, Take Three (4)

- Now the cache container implementation can almost fit into one slide…

```
class CacheContainer :
  private CacheContainerBranch<PointRecord,
          CacheContainerBranch<RectangleRecord,
          CacheContainerBranch<CircleRecord,
          CacheContainerBranch<LineRecord,
          CacheContainerBranch<PieRecord,
          CacheContainerBranch<TriangleRecord,
                               CacheContainerBase>>>>>>
{
  template<typename T>
  bool get(const Id_t id, T & record) const
  { return CacheContainerLeaf<T>::getCache().get(id, record); }

  template<typename T>
  void set(const T & record)
  { CacheContainerLeaf<T>::getCache().set(record); }

  template<typename T>
  size_t erase(const Id_t id)
  { return CacheContainerLeaf<T>::getCache().erase(id); }
  …
```

A Cache Container, Take Three (5)

- The erase functions are trivial:

```
  …
  size_t erase(const RecordTypeId_t recordTypeId, const Id_t id)
  {  return iteratingErase(recordTypeId, id); }

  size_t eraseAll(const Id_t id)
  { return iteratingEraseAll(id); }
};
```

A Cache Container, Take Three (5)

- There is almost no code duplication left. The problem has been solved at the template step on the Ladder of Evilness **except** for the template instantiation part:

```
class CacheContainer :
  private CacheContainerBranch<PointRecord,
          CacheContainerBranch<RectangleRecord,
          CacheContainerBranch<CircleRecord,
          CacheContainerBranch<LineRecord,
          CacheContainerBranch<PieRecord,
          CacheContainerBranch<TriangleRecord,
                               CacheContainerBase>>>>>>
```

- Whenever a new record type is added, we need to add it here and we most likely need to add the new type in some other places too. So the list of types is duplicated.
- In case we need to have two or more cache containers for different types (e.g. one for 2D geometry and one for 3D geometry), we need to write a cache container for each set of types.
- The CacheContainer could be a template that takes a list of types as template parameter.
- Let's create a template operation that can build this inheritance hierarchy.

What is a Typelist?

- The Typelist definition is quite simple:

```
template<typename HEAD, typename TAIL>
struct Typelist
{
  typedef HEAD Head;
  typedef TAIL Tail;
};

struct NullType { };
```

- **Head** is the type at the head of the list.
- **Tail** is the rest (or tail) of the list. Tail is either another typelist instantiation or NullType.
- The **NullType** is used for ending a typelist and is considered an empty typelist.
- Here is a typelist of an int, double, long and a std::string:

```
typedef Typelist<int,
        Typelist<double,
        Typelist<long,
        Typelist<std::string, NullType>>>> Types_t;
```

C++/Variadic Templates

- **Variadic Templates** are in the C++ 11 standard. With variadic templates it is possible to specify a variable number of template parameters:

```
typelist<int, double, long, std::string> Types_t;
```

- Alternatively like this:

```
GenerateTypelist<int, double, long, std::string>::Result Types_t;
```

- The syntax is much nicer.
- Variadic Templates are not supported in the compiler we use (VS 2012).
- I will talk about this on GOTO 2015. ☺

Template specialization

- Templates can be specialized for specific types:

```
template<typename T>
struct IsNullType
{
  enum { Result = false };
};

template<>
struct IsNullType<NullType>
{
  enum { Result = true };
};
```

- IsNullType<T>::Result = true if T = NullType.
- IsNullType<T>::Result = false for all other types.
- This is determined at compile time.

Parameterized Template specialization (1)

- Let's implement an IsTypelist template. The default implementation is straight forward:

```
template<typename T>
struct IsTypelist
{
  enum { Result = false };
};
```

- Template specializations can be parameterized.

```
template<typename HEAD, typename TAIL>
struct IsTypelist<Typelist<HEAD, TAIL>>
{
  enum { Result = true };
};
```

- IsTypelist<Typelist<T, U>>::Result = true for all types T and U.
- IsTypelist<T>::Result = false for all other types T.

Typelists of Records

- The available record types can be listed in two typelists of the records we support; one for 2D and one for 3D:

```
typedef Typelist<PointRecord,
        Typelist<RectangleRecord,
        Typelist<CircleRecord,
        Typelist<PieRecord,
        Typelist<LineRecord,
        NullType>>>>> Geometry2DTypelist;

typedef Typelist<Point3DRecord,
        Typelist<Box3DRecord,
        Typelist<Circle3DRecord,
        Typelist<Line3DRecord,
        NullType>>>> Geometry3DTypelist;
```

- We should have a typelist containing all supported types, so we merge (union):

```
typedef Merge<Geometry2DTypelist, Geometry3DTypelist>::Result ModelTypelist;
```

Typelist Operations

- Typelists can be manipulated by **typelist operations**.
- Example: **Merge** merges typelists LEFT and RIGHT **at compile time**. Duplicates are allowed.

```
template<typename LEFT, typename RIGHT>
struct Merge;

template<typename RIGHT>
struct Merge<NullType, RIGHT>
{
  typedef RIGHT Result;
};

template<typename HEAD, typename TAIL, typename RIGHT>
struct Merge<Typelist<HEAD, TAIL>, RIGHT>
{
  typedef Typelist<HEAD, typename Merge<TAIL, RIGHT>::Result> Result;
};
```

Merging Example

```
Merge<Typelist<U1, Typelist<U2, NullType>>,
        Typelist<U3, NullType>>::Result =
Typelist<U1,
  Merge<Typelist<U2, NullType>, Typelist<U3, NullType>>::Result> =
Typelist<U1,
  Typelist<U2,
    Merge<NullType, Typelist<U3, NullType>>::Result> =
Typelist<U1,
  Typelist<U2,
            Typelist<U3, NullType>>> =
Typelist<U1, Typelist<U2, Typelist<U3, NullType>>>
```

Generic Compile Time Recursion of a Typelist

- The **ApplyRecursive** typelist operation will do compile time recursion of a typelist.
- The template takes three template parameters:
  - A template that takes a type T and a type APPLIED_TAIL as template parameters.
  - A type BASE.
  - A typelist TLIST containing the types the template should be applied to.
- NOTE: The template TEMPLATE is a **template template** parameter of the template ApplyRecursive.
- In the NullType case, the result is BASE.
- In the Typelist<HEAD, TAIL> case, the result is the template TEMPLATE instantiated with T = HEAD and APPLIED_TAIL = ApplyRecursive of the tail.

```
template<template<typename T, typename APPLIED_TAIL> class TEMPLATE,
         typename BASE,
         typename TLIST>
struct ApplyRecursive;
```

ApplyRecursive implementation

```
template<template<typename T, typename APPLIED_TAIL> class TEMPLATE,
         typename BASE>
struct ApplyRecursive<TEMPLATE, BASE, NullType>
{
  typedef BASE Result;
};


template<template<typename T, typename APPLIED_TAIL> class TEMPLATE,
         typename BASE,
         typename HEAD, typename TAIL>
struct ApplyRecursive<TEMPLATE, BASE, Typelist<HEAD, TAIL>>
{
private:
  typedef typename ApplyRecursive<TEMPLATE, BASE, TAIL>::Result AppliedTail;
public:
  typedef TEMPLATE<HEAD, AppliedTail> Result;
};
```

- The **ApplyRecursive** typelist operation is a simplified version of the **GenScatterHierarchy** typelist operation described in Alexandrescus book.

ApplyRecursive Example

```
ApplyRecursive<T, B, Typelist<U1, Typelist<U2, Typelist<U3, NullType>>>>::Result =
T<U1,
   ApplyRecursive<T, B, Typelist<U2, Typelist<U3, NullType>>>::Result> =
T<U1,
 T<U2,
    ApplyRecursive<T, B, Typelist<U3, NullType>>::Result>> =
T<U1,
 T<U2,
  T<U3,
     ApplyRecursive<T, B, NullType>::Result>>> =
T<U1,  T<U2, T<U3, B>>>
```

A Cache Container, Final (1)

- Let's take yet another step down the ladder to template metaprogramming.
- Let's use ApplyRecursive to build the inheritance hierarchy:

```
class CacheContainer :
  private CacheContainerBranch<PointRecord,
          CacheContainerBranch<RectangleRecord,
          CacheContainerBranch<CircleRecord,
          CacheContainerBranch<LineRecord,
          CacheContainerBranch<PieRecord,
          CacheContainerBranch<TriangleRecord,
                               CacheContainerBase>>>>>>
```

- Please recall:

```
template<typename T, typename BASE>
struct CacheContainerBranch;

template<template<typename T, typename APPLIED_TAIL> class TEMPLATE,
         typename BASE,
         typename TLIST>
struct ApplyRecursive;
```

- ApplyRecursive can create the inheritance hierarchy by using CacheContainerBranch as TEMPLATE, CacheContainerBase as BASE and e.g. Modeltypelist as TLIST.

A Cache Container, Final (2)

- So here is the CacheContainer implemented as a class template that will instantiate the cache for all types in the typelist TLIST:

```
template<typename TLIST>
class CacheContainer
  : private ApplyRecursive<CacheContainerBranch,
                           CacheContainerBase,
                           TLIST>::Result
{
public:
  template<typename T>
  bool get(const Id_t id, T & record) const
  { return CacheContainerLeaf<T>::getCache().get(id, record); }

  template<typename T>
  void set(const T & record)
  { CacheContainerLeaf<T>::getCache().set(record); }

  template<typename T>
  size_t erase(const Id_t id)
  { return CacheContainerLeaf<T>::getCache().erase(id); }
  …
```

A Cache Container, Final (3)

```
  …
   size_t erase(const RecordTypeId_t recordTypeId, const Id_t id)
   {  return iteratingErase(recordTypeId, id); }

   size_t eraseAll(const Id_t id) { return iteratingEraseAll(id); }
};
```

- And finally an instantiation:

```
CacheContainer<ModelTypelist> cacheContainer;
```

- Now we are done.
  - New types can be added without requiring changes to the cache container.
  - The CacheContainer can be reused for multiple lists of types.

A Cache Container, Final (4)

- Whenever a new data package is created, the record type is added to the ModelTypelist typelist and is automatically supported by the cache container.
- Adding a new type has become declarative.
- The cache container has gone from requiring tedious, repeated, error prone, manual and duplicated work to being maintenance free (in regards to adding new types).
- The CacheContainer developer does not need to implement the compile time recursion to build the inheritance hierarchy. The **ApplyRecursive** typelist operation does that.
- This is where you should burst into tears of joy!

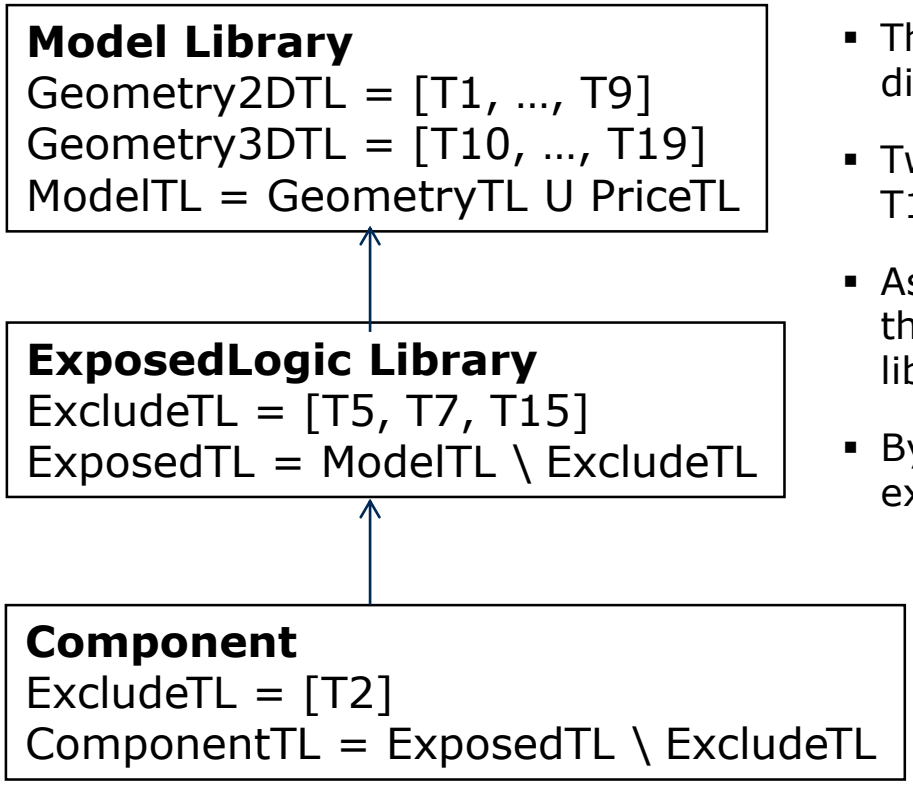
The ApplyRecursive Possibilities

- The template passed as template parameter to ApplyRecursive can inherit from the applied tail and thereby get access to the instantiation of the class and has get access to all protected and public members and functions.
- The template passed as template parameter to ApplyRecursive does not have to inherit from the applied tail. In this case the it has access to the class of the inherited tail, i.e. type information, public or protected static members and static member functions.
- ApplyRecursive allows us to do **runtime iteration** over a list of types by doing **compile time recursion**.
- The compiler will be busy. Expect some very long compile times for large typelists and template instantiations that will generate a lot of code.
- Consider using pImpl to avoid #includes in the header file.

Typelist Operations

- Template specialization and compile time recursion enables us to iterate through the types in a typelist at runtime.
- Recursive typelist iteration can be executed by typelist operations. The developer does not have to know how to do compile time recursion.
- Typelists can be manipulated by **Typelist Operations** that are template classes that can do specific operations, e.g.:
  - Set union, intersection and complement.
  - Take a typelist as input and return the typelist with duplicates removed.
  - Apply a template to all members of a typelist.

Use of Typelists and Typelist Operations in multiple Libraries



- The shared Model library contains a number of distinct record types.
- Two typelists of the types (T1..T9 and T10..T19) are defined and merged into one.
- As new types are added to the Model library they can automatically be supported by the libraries/components using it.
- By using typelist operations, types can be excluded if they are not needed.

SAXO BANK

ApplyRecursive and Merge

- ApplyRecursive and Merge follow the same pattern:

```
template<typename LEFT, typename RIGHT>
struct Merge
{
  typedef typename ApplyRecursive<Typelist, RIGHT, LEFT>::Result Result;
};
```

Merge<Typelist<U1, Typelist<U2, NullType>>, Typelist<U3, NullType>>::Result =

ApplyRecursive<Typelist, Typelist<U3, NullType>,

Typelist<U1, Typelist<U2, NullType>>>::Result =

Typelist<U1,

ApplyRecursive<Typelist, Typelist<U3, NullType>, Typelist<U2, NullType>>::Result> =

Typelist<U1, Typelist<U2,

ApplyRecursive<Typelist, Typelist<U3, NullType>, NullType>::Result>> =

Typelist<U1, Typelist<U2, Typelist<U3, NullType>>>

Using the cache container (brag sheet)

- The cache container is wrapped in a COM component.
- The cache container is implemented using pImpl pattern so that the compile time hit is taken when compiling one .cpp file only.
- There is one COM interface for each type. There are 10+ functions for each type.
- The COM interfaces are generated in C++ using embedded IDL and macros.
- The COM object factory iterates through a typelist to find the right interface to instantiate.
- The test of the COM component is done by a template that can test all interface.
- The test template is instantiated and executed for all types using a typelist.
- There are fewer lines of code in the test program than there are COM interface functions being tested.
- When a new record type is added to the typelist in the model library, the library can be linked and the component and the component will automatically have a new interface for the type (except for the GUID of the interface).

Other Examples

- In Saxo Bank we have also used typelists for:
  - Streaming the data packages we support to different formats.
  - Generating database schemas.
- We have used **macro metaprogramming** for:
  - Generating constants. The same constants are needed in the model library as well as in several COM and managed interfaces. A constant generating macro is created that can be used for generating constants in different shapes.
- We have used automated code generation for:
  - Generating managed versions of the data packages including conversion functions, unit tests and performance tests.

Summary

- “Use the right hammer” i.e. pick the right step on the Ladder of Evilness.
- Typelists and typelist operations can be used where the same template should be instantiated for a large number of types.
- Typelists and typelist operations enables us to iterate through the types in a typelist at runtime.
- Trivial code can be generated by the compiler instead of being created manually.
- In certain scenarios the use of templates, typelists and typelist operations can replace trivial work with maintenance free code (in regards to supporting new types).

Q&A

Questions are also more than welcome by e-mail:

jcl@saxobank.com / jcl@jespergaerde.dk

SAXO BANK

Thanks!

- Thanks for attending.
- Special thanks to my colleagues in Saxo Bank for giving constructive feedback.
- Thanks to Saxo Bank for letting me do the talk and allowing me to spend time on the presentation.

SAXO BANK